Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Juha Heikkilä

Opinnäytetyö

Model-View-Controller-arkkitehtuuri mobiiliohjelmoinnissa
Case: Matka- ja kululasku

Työn Ohjaaja: Maritta Hoffren, FM, lehtori
Työn Tilaaja: Kilosoft Oy, Talousjohtaja Kimmo Hakkarainen
Tampere 12/2009

Tampereen Ammattikorkeakoulu, Tietojenkäsittelyn koulutusohjelma,
Ohjelmistotuotanto

Työn tekijä	Juha Heikkilä
Työn nimi	Model-View-Controller Arkkitehtuuri Mobiiliohjelmoinnissa Case: Matka- ja Kululasku
Sivumäärä	31
Valmistumisaika	joulukuu 2009
Työn ohjaaja	Maritta Hoffrèn, FM, lehtori
Työn tilaaja	Kilosoft Oy, Talousjohtaja Kimmo Hakkarainen

TIIVISTELMÄ

Opinnäytetyön toimeksiantajana on tamperelainen ohjelmistoyritys, Kilosoft Oy. Toimeksiantajayritys on ohjelmistokehityksen, laadunvarmistuksen ja verkonhallintajärjestelmien asiantuntijayritys.

Toimeksiantaja toivoi, että rakennettavalla mobiilisovelluksella saataisiin helpotusta paperisten matka- ja kululaskujen hoitamiseen. Ajatuksena oli, että työntekijä voisi jo työmatkan aikana kirjata matkan aiheuttamat matka- ja kululaskut.

Sovelluksen arkkitehtuurissa otettiin huomioon, että sovellus ei irrallisena tuotteena ole kovin kaupallinen työkalu, joten sovelluksesta tuli tehdä helposti muihin sovelluksiin integroitava. Tästä syystä sovellus suunniteltiin käyttämään Model-View-Controller-arkkitehtuuria.

Opinnäytetyötä luettaessa olisi hyvä olla perustiedot oliopohjaisesta ohjelmoinnista, mobiiliohjelmoinnista sekä ohjelmistoprosessista. Nämä eivät ole välttämättömiä, mutta niiden tuntemus auttaa hahmottamaan työtä paremmin.

Opinnäytetyö käsittelee MVC-arkkitehtuurin soveltuvuutta sekä mobiilisovelluksissa että yleisesti.Työn tuloksena todetaan, että MVC-arkkitehtuuri ei sovellu parhaiten mobiilipuolen ohjelmistojen kehittämiseen. Se on kuitenkin varteenotettava vaihtoehto joissakin tapauksissa, kuten sovelluksissa missä Malli-osuus pohjautuu verkon ylitse käytettävään tietokantaan. Työn tuloksia voidaan hyödyntää tulevien ohjelmistoprojektien arkkitehtuurivalinnoissa.

Avainsanat MVC, Malli, Näkymä, Ohjain, Arkkitehtuuri, Mobiiliohjelmointi

Writer(s)	Juha Heikkilä
Thesis	Model-View-Controller Architecture in Mobile Programming Case: Mobile Traveling Expenses
Pages	31
Graduation time	December 2009
Thesis Supervisor	Maritta Hoffrèn, FM, lecturer
Co-operating Company	Kilosoft Oy, Finance Director Kimmo Hakkarainen

ABSTRACT

The client for the Bachelor's thesis work was Kilosoft Oy, a company that specializes in software development, quality verification (testing) and network management. The company originates in Tampere.

Kilosoft Oy requested software to replace the old paper version of travelling-expenses account with an easy-to-use mobile software that would enable the user to fill out the required data while traveling, thus saving time and effort.

In planning of the project, it was taken into consideration that such a software would not be very marketable as a stand-alone application, so the architecture was to be made in a way which would allow easy integration into other software packages. As a result, MVC-architecture was selected.

Basic knowledge of object-oriented programming, mobile programming and general software development process will help while reading this documentation. However, none of them are required to understand this thesis work.

This work deals mainly with adapting MVC-architecture into mobile software development and as a result, we will find out that MVC-architecture might not be the best option for the most common mobile applications. The results of this thesis work can be used as a guide while selecting software architecture in future development projects.

Keywords MVC, Architecture, Mobile programming, Model, View, Controller

Sisällysluettelo

Käytetyt termit

J2ME	Java 2 Micro Edition – Java luokkakirjastot mobiiliohjelmointiin
CLDC	Connected Limited Device Configuration – osa J2ME ohjelmistokehystä tarkoitettu rajoitettuja resursseja sisältävien laitteiden sovelluksille
MIDP	Mobile Information Device Profile- Osa J2ME ohjelmistokehystä
MVC	Model – View – Controller – Trygve Reenskaugin kehittämä Ohjelmistokehitysmalli
PARC	Palo Alto Research Center – Tutkimuskeskus joka tunnetaan parhaiten graafiseen käyttöliittymään liittyvästä kehitystyöstä. Perustettu alunperin Xeroxin tutkimusyksiköksi.
IMSI	International Mobile Subscriber Identity – SIM-kortille tallennettu ID jonka avulla voidaan yksilöidä jokainen matkapuhelinverkon käyttäjä.
IMEI	International Mobile Equipment Identity – Matkapuhelimen laitetunnus jonka avulla voidaan yksilöidä jokainen matkapuhelin verkosta.
API	Application Programmer Interface – Ohjelmointialustojen sekä eri ohjelmistojen tarjoama rajapinta jonka kautta päästään käsiksi ohjelman sisäisiin toimintoihin.
SOAP	Simple Object Access Protocol – Microsoftin kehittämä, World Wide Web Consortiumin ylläpitämä standardi jonka päätehtävä on mahdollistaa proseduurien etäkutsuja. SOAP pohjautuu XML-kieleen ja mahdollistaa toiminnan useiden eri protokollien ylitse

1 Johdanto

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimii Kilosoft Oy, joka on vuonna 2002 perustettu ohjelmistokehitykseen, laadunvarmistukseen ja verkonhallintajärjestelmiin keskittynyt asiantuntijayritys. Kilosoftilla palveluksessa on tällä hetkellä noin 50 työntekijää. Suomessa Kilosoft Oy toimii Espoossa, Tampereella ja Jyväskylässä. Kansainvälinen toiminta tapahtuu yhteistyökumppaneiden kautta.

Kilosoft Oy:n pääasiallinen liiketoiminta on henkilöstövuokraus sekä projektitoimitukset. Lisäksi Kilosoft Oy:n liiketoimintaan kuuluu verkonhallintajärjestelmien suunnittelu ja toteutus käyttäen Kilosoft Oy:n kehittämää NetWrapper-teknologiaa sekä tietoturvaraportointipalvelut Kilosoft Oy:n FiERS-teknologian avulla.

1.2 Tavoitteet

Opinnäytetyön tavoitteena on toteuttaa ohjelmistoprojektina matkapuhelimelle Matka- ja kululaskuohjelmisto Java-ohjelmointikieltä käyttäen sekä henkilökohtaisesti perehdyttää itseäni MVC-arkkitehtuurin käyttöön ja parantaa taitojani mobiiliohjelmoinnissa sekä syventää tietämystäni arkkitehtuurisuunnittelusta.

Sovelluksesta oli tarkoitus tehdä helposti integroitava ja sovelluskoodista uudelleenkäytettävää. Tätä tarkoitusta varten sovellusarkkitehtuuriksi valittiin Model-View-Controller arkkitehtuuri. Lisäksi otettiin mukaan Observer-malli, tavoitteena nopeuttaa ohjelman toimintaa.

1.3 Rajaus

Opinnäytetyö keskittyy Model-View-Controller arkkitehtuuriin mobiiliohjelmiston näkökulmasta. Tässä työssä kerrotaan yleisesti MVC-arkkitehtuurin toimivuudesta ja käyttötarkoituksista sekä Matka- ja kululasku-sovelluksen näkökulmasta. Opinnäytteessä sivutaan myös MVC-arkkitehtuurin merkitystä ohjelmistotestauksessa.

1.4 Keskeinen kirjallisuus

Opinnäytetyöhön kirjallisuutta löytyi paljon, mutta kaikki kirjallisuus käsitteli itse MVC-arkkitehtuuria hyvin suppeasti. Tämän vuoksi opinnäytteeni sisältö pohjautuu suurimmaksi osaksi ohjelmointiprojektin kautta tulleeseen kokemukseen.

2 MVC-Arkkitehtuuri

2.1 Historia

MVC-arkkitehtuurin kehitti norjalainen Trygve Reenskaug vuonna 1979 työskennellessään Smalltalk-ohjelmointikielen kehityksen parissa Xerox-PARC tutkimuskeskuksessa. Alkuperäisessä MVC-arkkitehtuurissa oli Mallin, Näkymän ja Ohjaimen lisäksi myös Editori (Editor) joka oli tarkoitettu tilapäiseksi komponentiksi Näkymän ja syötelaitteen (hiiri, näppäimistö) välille. (Origins of Software Architecture Study 2008 [online] [8.12.2008])

2.2 Arkkitehtuuri

Arkkitehtuurisuunnittelun tarkoituksena on mahdollistaa sovellukseen kehitettäville komponenteille paras mahdollinen rakenne. (Aalto, Aalto, Jaaksi, Vättö, 1999, 40) Ohjelmistoarkkitehtuurit on suunniteltu nopeuttamaan ja helpottamaan ohjelmistojen kehitystä tarjoamalla niihin soveltuvia testattuja ja toimiviksi todettuja ohjelmointimenetelmiä. Näitä menetelmiä on useita, mutta tässä työssä tarkastelen Model-View-Controller- sekä Observer-arkkitehtuurimalleja.

2.3 Arkkitehtuurin valitseminen

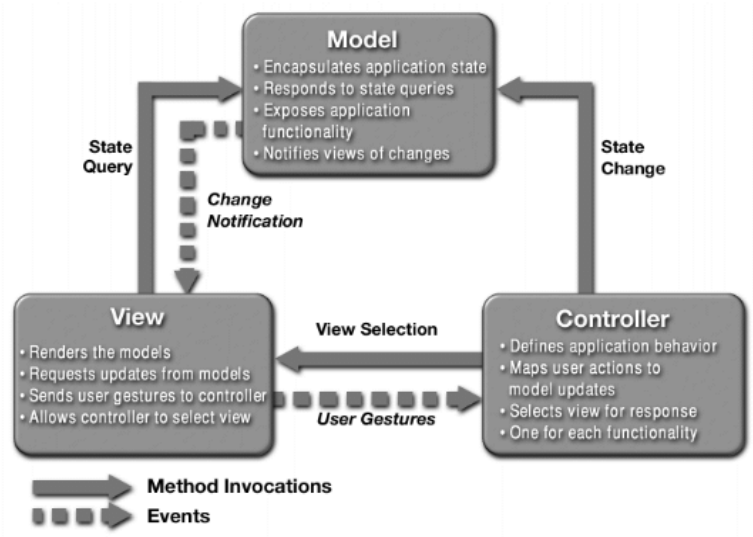
Mikäli ohjelmistoprojektissa aiotaan käyttää jotain valmista arkkitehtuurisuunnitelmaa, on arkkitehtuurin oikea valinta oleellinen osa kehitysprosessia ja sopivan arkkitehtuurin valintaan kannattaa panostaa resursseja. Vaikka arkkitehtuurit on suunniteltu nopeuttamaan ja helpottamaan ohjelmistotuotantoa, ne saattavat tuoda mukanaan myös joitakin ei toivottuja ominaisuuksia, jotka voivat haitata projektin sujuvaa etenemistä.

Pahimmassa tapauksessa huonosti valitun arkkitehtuurin tuomat ongelmat ilmenevät vasta siinä vaiheessa kehitysprosessia, kun arkkitehtuurin muuttaminen on jo miltei mahdotonta tai vaihdosta aiheutuisi vähintäänkin kohtuuttomia ylimääräisiä kuluja.

2.4 Yleistä MVC-arkkitehtuurista

MVC-arkkitehtuurin perusajatus on rikkoa yksittäinen komponentti kolmeen erilliseen osaan: Malliin, Näkymään ja Ohjaimeen (Kuva 1). Tätä kaavaa noudattaen pystytään erottamaan käyttöliittymä sovelluslogiikasta ja luomaan helposti ylläpidettäviä komponentteja (Koskimies & Mikkonen, 2005, 142).

MVC-arkkitehtuuria noudattamalla saadaan ohjelmistokomponentit helposti vaihdettaviksi ja tämä nopeuttaa erillisistä osista tehtyjen ohjelmistokokonaisuuksien kokoamista sekä helpottaa niiden räätälöimistä eri käyttötarkoituksiin soveltuviksi.



Kuva 1: MVC-Arkkitehtuurikuvaus (Java BluePrints - J2EE Patterns – Model-View-Controller)

Kuten kuvasta 1 on nähtävissä, MVC-arkkitehtuurin toiminta perustuu kolmen luokan yhteistoimintaan. Tässä toimintamallissa Malli-komponentti toimii teoriassa yksinäisenä komponenttina, jonka tiedot Näkymä muuttaa käyttäjälle tulkittavaksi. Ohjain-komponentti saa Näkymä-komponentilta käyttäjän tekemät valinnat ja sen pohjalta tulkitsee tiedon Mallille, joka reagoi ohjelmoidulla tavalla.

Malli-olio voi esimerkiksi päivittää tietojansa käyttäjän antamien syötteiden mukaisesti ja suorittaa toimintoja tiedon muokkaamiseksi itsensä kannalta oleelliseen muotoon. Tämän jälkeen Ohjain valitsee, mikä Näkymä käyttäjälle seuraavaksi näytetään.

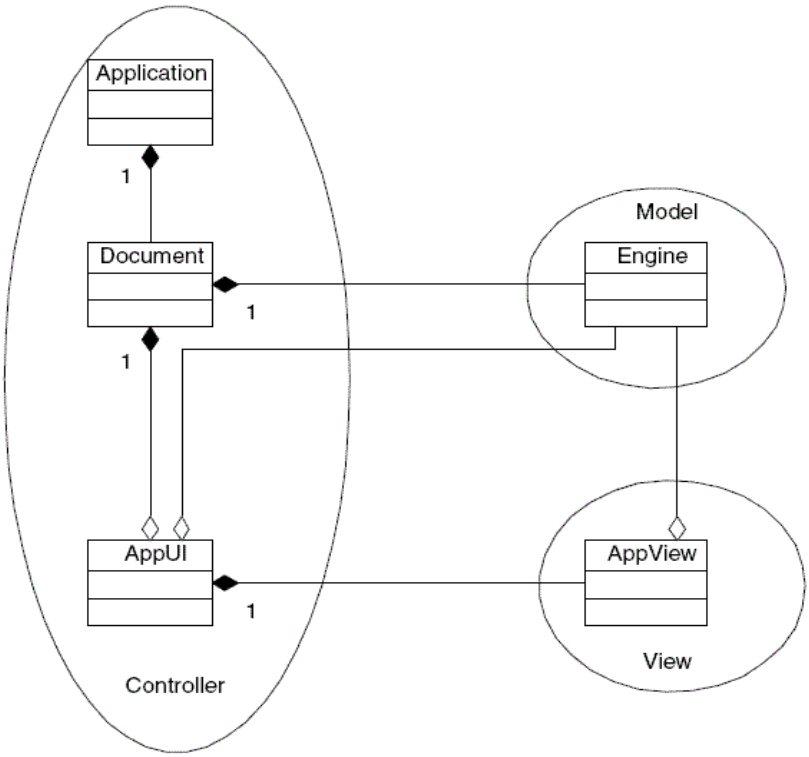
Ohjain- ja Näkymä-luokkien on hyvä toteuttaa Tarkkailija-rajapinta. Näin ne voivat rekisteröityä tarvitsemilleen Malli-luokille ja ilmoittaa tätä kautta haluavansa tiedon Malli-luokkaan kohdistuvista päivityksistä. Tarkkailija-rajapintansa kautta Ohjain- ja Näkymä-luokat saavat jatkuvasti tiedon Malli-luokissa tapahtuvista muutoksista ja osaavat päivittää tietonsa ajantasaisiksi. (Koskimies & Mikkonen, 2005, 142).

MVC-arkkitehtuuri soveltuu graafisen käyttöliittymän omaavien ohjelmistojen tekemiseen hyvin. Hajauttamalla näkymä ja tieto toisistaan erillisiksi komponenteiksi, voimme muokata erilaisia graafisia käyttöliittymiä siten, ettei muita komponentteja tarvitse muokata. Useampi näkymä voi myös hyväksikäyttää samoja komponentteja mahdollistaen tiedon esittämisen käyttäjäspesifistisenä tietona ja helpottaen ohjelmiston lokalisointia. Tämä tekee arkkitehtuurista luonnollisen valinnan graafista käyttöliittymää käyttäviin sovelluksiin (Koskimies & Mikkonen, 2005, 144).

Joissakin tapauksissa Näkymä- ja Ohjain-luokka voidaan yhdistää yksittäiseksi komponentiksi. Tämä on harkittavissa oleva ratkaisu silloin, kun näkymällä ei ole paljoa toimintaa eikä sille siten ole järkevää toteuttaa omaa Ohjain-luokkaa. (Koskimies & Mikkonen, 2005, 144)

Kovin yksinkertaisiin sovelluksiin MVC-arkkitehtuuri ei sovi, sillä arkkitehtuurin rakenne monimutkaistaa monia yksinkertaisia asioita (Koskimies & Mikkonen, 2005, 144). MVC-arkkitehtuuri soveltuu paremmin kompleksisiin ohjelmistoihin, joissa on useita näkymiä sekä tietomalleja.

Johtuen ohjelmointialustan rakenteesta, on Symbian S60 -ohjelmointialustalla MVC-arkkitehtuurin käyttö miltei pakollista. Tämä alusta pohjautuu Application-, Document-, AppUI- ja AppView -luokkien toteuttamiseen. Näistä luokista saadaan koottua aplikaatiolle Ohjain sekä Näkymä. Pakollisten luokkien lisäksi rakennetaan sovellukselle Engine-luokka, joka on idealtaan jo sama kuin Malli-luokka. (Kuva 2)



Kuva 2: MVC-Arkkitehtuuri Symbian alustalla (Design Patterns in Symbian 2007. [online] [viitattu 26.11.2008])

2.5 Model (Malli)

Malli kuvaa osaa tai kokonaisuutta sovelluksen tiedoista ja tarjoaa sovellukselle näkymien tarvitsemaa tietoa sekä pitää huolen tiedon oikeellisuudesta. (Koskimies & Mikkonen, 2005, 143-144) Tämä luokka ei omista tietoa ohjaimestaan eikä näkymistään. Mikäli käytetään Tarkkailija-arkkitehtuuria MVC-arkkitehtuurin lisäksi, omistaa Malli epäsuoran yhteyden Näkymiinsä. Malli-luokka ilmentää liikelogiikan sekä -tiedon, ja hallinnoi tiedon päivitykset ja pääsyn tietoihin. (Koskimies & Mikkonen, 2005, 142) Malli-luokka voi toteuttaa Tarkkailija-rajapinnan ja ilmoittaa tätä kautta päivittyneistä tiedoistaan niille Näkymä-luokille, jotka ovat rekisteröityneet kuuntelijoiksi. (Koskimies & Mikkonen, 2005, 142)

2.6 View (Näkymä)

Näkymän tarkoitus MVC-arkkitehtuurissa on edustaa graafista käyttöliittymää (Koskimies & Mikkonen, 2005, 142). Tämä siis vastaanottaa käyttäjän komennot sekä näyttää käyttäjälle sillä hetkellä oleelliset tiedot Mallista. Näkymä on vastuussa tiedon näyttämisestä ja huolehtii siitä, että tiedot ovat aina ajantasaisia. Tietojen ajantasaisuutta Näkymä pystyy ylläpitämään kyselemällä Mallilta uudelleen tiedot aina kun niitä tarvitaan.

Mikäli Tarkkailija-arkkitehtuuria käytetään MVC-arkkitehtuurin lisäksi, voi Näkymä rekisteröidä itsensä kuuntelijaksi tarvitsemalleen Mallille. Näin Näkymästä tulee laitteistoystävällisempi, koska sen ei tarvitse kysellä päivitettyjä tietoja jokaisella piirtokierroksellaan vaan se voi luottaa Malli-luokan ilmoittavan muutoksistaan sitä mukaa kun niitä tulee. (Java BluePrints – J2EE Patterns [online] [8.12.2008])

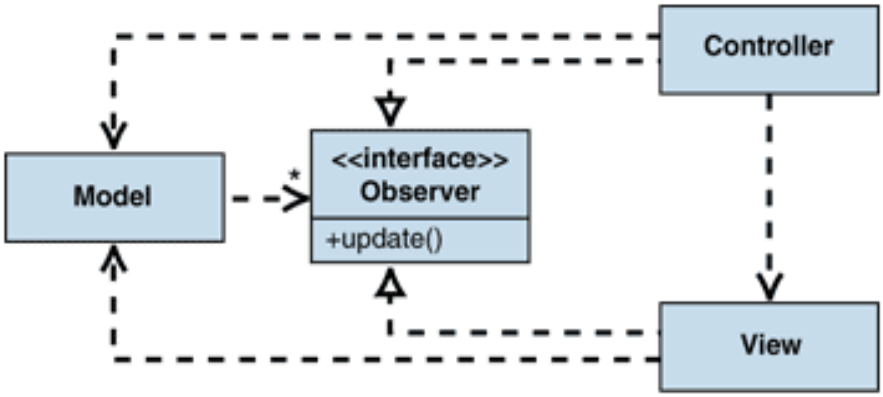
2.7 Controller (Ohjain)

Ohjain-luokka ohjaa Näkymä-luokan toiminnallisuutta sekä välittää tietoja Malli- ja Näkymä-luokkien välillä toimien sovittimen sovittimena. (Koskimies & Mikkonen, 2005, 142) Ohjain-luokan toimintoihin kuuluu ohjata käyttäjän tekemät valinnat ja muutokset Malli-luokan tiedoksi, sekä valita oikea Näkymä-luokka tilanteen mukaisesti. (Java BluePrints – J2EE Patterns [online] [8.12.2008]) Tämä luokka on tietoinen kaikista mahdollisista toiminnoista, mitkä käyttäjän sallitaan suorittaa. (Yuan 2004, 89)

2.8 Observer (Tarkkailija)

Vaikka Tarkkailija ei virallisesti ole osa MVC-arkkitehtuuria, sen käyttö on suositeltavaa MVC-arkkitehtuurin kanssa. Tarkkailija-arkkitehtuuria hyväksikäyttäen Näkymä-luokat eivät joudu kyselemään Malli-luokilta jatkuvasti muuttuneita tietoja, vaan ne voivat pitää omaamansa tiedot Näkymä-luokassaan kunnes Malli-luokka

ilmoittaa niille saaneensa muuttunutta informaatiota. Mikäli Näkymä-luokka ei ole kuuntelemassa Malli-luokkansa muutoksia, se joutuu kyselemään informaationsa Mallilta aina uudelleen jokaisella piirroskierroksella ja tämä saattaa hidastaa ohjelmakoodin suoritusta rankastikin. Tarkkailija-rajapinta sijoittuu MVC-arkkitehtuuriin kuvan 3 mukaisesti.



Kuva 3: Tarkkailija-rajapinnan käyttö Näkymän ja Ohjaimen erottamiseen Mallista. (http://msdn.microsoft.com/en-us/library/ms978748.aspx [online 26.11.2008])

2.9 MVC-arkkitehtuuri – hyödyt

Model-View-Controller arkkitehtuurin suurimpana hyötynä pidetään sen tuomaa modulaarisuutta, joka helpottaa komponenttien käyttämistä toisistaan erillisinä ohjelman osina. Tämän lisäksi arkkitehtuuri antaa mahdollisuuden kehittäjille paneutua oman erikoisosaamisensa mukaisesti ohjelmiston kehittämiseen. Käyttöliittymäsuunnittelijat voivat luoda Näkymät, tietokantasuunnittelijoille voidaan jättää Mallien suunnittelu ja Ohjaimen suunnittelemisesta voi vastata järjestelmäsuunnittelijat. Koska opinnäytetyössä pohjana käytettyä sovellusta kirjoitettaessa ei luonnollisestikaan hyödynnetty erikoisosaajien apua, auttoi MVC-arkkitehtuuri ymmärtämään ja keskittämään toiminnallisuuden oikeisiin komponentteihin. Tämän ohjelmakokonaisuuden hajoittaminen teki ohjelmakoodista helpommin ylläpidettävän kokonaisuuden, jossa kukin toiminnallisuus on keskitetty omiin luokkiinsa.

2.10 MVC-arkkitehtuuri - haitat

MVC-arkkitehtuurin suurimpana haittana pidetään sen tuomaa ylimääräistä ohjelmiston monimutkaisuutta. Arkkitehtuurin pakottama kolmen luokan malli lisää ohjelmistoon kompleksisuutta, jolta ainakin pienemmmissä sovelluksissa pystyttäisiin monesti välttymään käyttäen yhtä luokkaa kaikkien arkkitehtuurimallin luokkien sijaan.

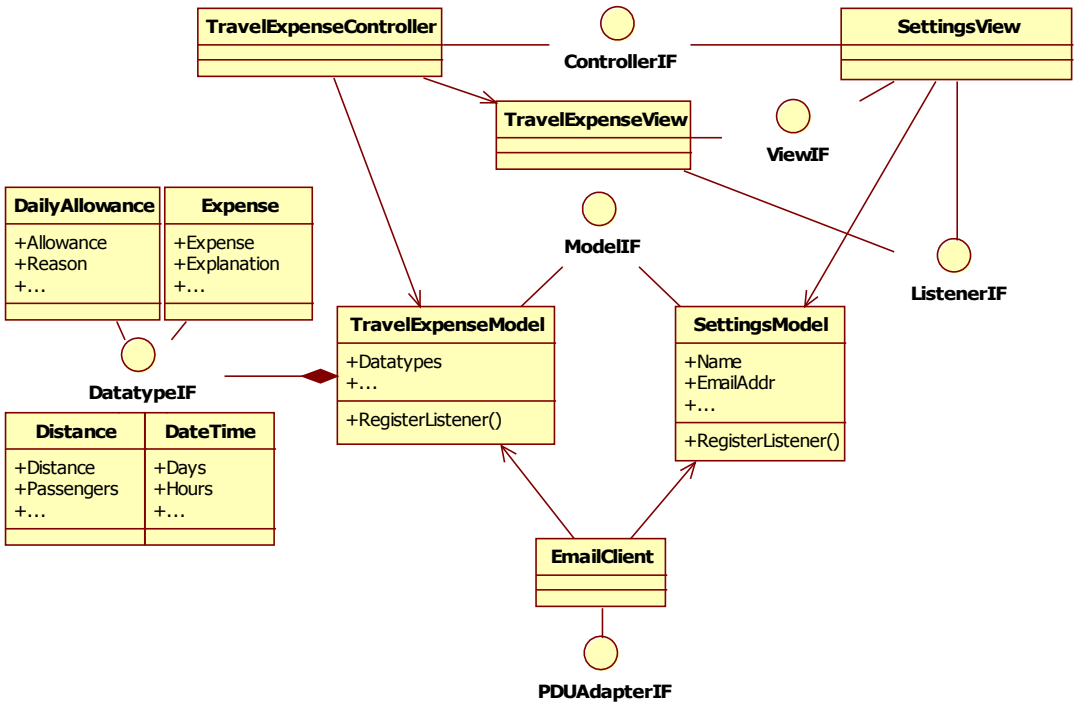
Järjestelmän rakenne hidastaa jonkin verran käyttöliittymän piirtämistä, koska Näkymä joutuu kyselemään Mallilta tietoja jatkuvasti. Tämä luo ylimääräistä prosessorikuormitusta metodikutsujen myötä ja hidastaa ohjelmiston toimintaa. Normaaleissa tietokoneissa tästä ei suurta haittaa ole, mutta pienemmissä mobiililaitteissa se saattaa olla kriittinen tekijä toimivan ohjelman ja täydellisen fiaskon välillä.

Metodikutsujen aiheuttaman prosessorikuormituksen lisäksi MVC-arkkitehtuuri aiheuttaa suurempaa muistinkulutusta ylimääräisten luokkien muodossa. Tämäkään ei pöytätietokoneissa muodostu ongelmaksi, mutta esimerkiksi osa matkapuhelimista sisältää hyvinkin rajatun määrän tallennuskapasiteettia. Mobiililaitteet myös usein rajaavat kuinka paljon yksittäinen sovellus saa varata muistia.

3 Toteutus

Ohjelmisto toteutettiin Java -ohjelmointikielellä, noudattaen Malli-Näkymä-Ohjain-arkkitehtuuria ja hyväksikäyttäen Tarkkailija-rajapintaa.

Toteutuksessa (Kuva 4) on käytetty sekä täysin erillisiä Malli-, Näkymä- ja Ohjain-luokkia (Matka- ja kululasku) että luokkaa jossa Ohjain- ja Näkymä-luokat on yhdistetty (asetukset). MVC-arkkitehtuuri mahdollistaa Ohjain- ja Näkymä-luokkien yhdistämisen silloin, kun se on järkevää toteutuksen kannalta. Matka- ja kululaskuohjelmistossa tehdyssä Asetukset-luokassa tämä oli ideaallinen vaihtoehto. Kyseinen luokka oli riittävän yksinkertainen toteuttaakseen kummatkin rajapinnat itsenäisesti ilman, että sovelluskoodi olisi paisunut vaikeasti luettavaksi ja ylläpidettäväksi.



Kuva 4: Matka- ja Kululasku sovellus

3.1 Lähtökohdat

Matka- ja kululaskuohjelmiston kehittäminen lähti ideasta korvata ylimääräinen paperityö usein matka- ja kululaskuja kirjoittavilta henkilöiltä, nopeuttaen prosessia ja helpottaen laskujen tekemistä. Matka- ja kululaskuohjelmisto aloitettiin tyhjältä pöydältä. Aluksi keskustelimme Kilosoft Oy:n teknologiapäällikön kanssa projektista ja yhdessä toteutimme ohjelmistomäärittelyn sekä priorisoinnin. Projektin vaatimusmäärittelyssä sovittiin, mitkä toiminnot prototyypiltä vaaditaan ja mitkä ovat valinnaisia toiminnallisuuksia. Pakollisiin toiminnallisuuksiin kuuluivat muun muassa mahdollisuus syöttää normaaleihin matka- ja kululaskuun liittyviä menoeriä, esimerkiksi kilometrit, päivärahat ja muut kulut sekä mahdollisuus poistua sovelluksesta siten, että sovellus tallettaa jo syötetyt tiedot eikä käyttäjä joudu näitä syöttämään uudestaan. Tietojen poistamiseen sovelluksen muistista sovittiin hetki, jolloin sovellus on onnistuneesti lähettänyt tiedot sähköpostilla eteenpäin. Vaatimusmäärittelyssä sovittiin myös sekundäärisistä toiminnoista, jotka toteutetaan mikäli aikarajat ne sallivat. Näitä toimintoja oli mahdollisuus syöttää matkustajalukumäärä, kilometrikorvausten eri asteet: oma auto, autoetu, sivuvaunu, polkupyörä.

3.2 Työkalut

Ohjelmisto toteutettiin JavaME ohjelmointiympäristössä, käyttäen CLDC 1.0- ja MIDP 2.0 luokkakirjastoja. JavaME valittiin ohjelmointikieleksi pääasiassa sen takia, että Java ohjelmat ovat helposti siirrettävissä ja useat mobiililaitteet tukevat Java-ohjelmointikieltä nykyisin. Toinen peruste J2ME:n valintaan oli se, että itselläni on enemmän kokemusta kyseisellä kielellä ohjelmoimisesta ja projekti etenisi sujuvammin tällä, kuin esimerkiksi C++ pohjaisesti Symbian alustalla.

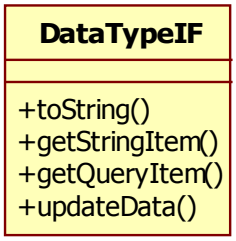
3.3 Rajapinnat

Ohjelmassa on käytetty kuutta erillistä rajapintaa, näistä kolme toimii MVC-arkkitehtuuria tukevana, yhden avulla toteutetaan Tarkkailijia-arkkitehtuuri ja kaksi toimii rajapintoina verkkoliikenteen ja käyttäjäsyötteen vastaanottajina.

3.3.1 ControllerIF

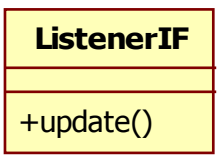
Tämä rajapinta ilmentää Ohjain-luokkaa. Vaikka Ohjain- ja Näkymä-luokat ovatkin kiinteästi toisissaan kiinni, on tämä rajapinta tehty päätason ohjainta helpottamaan. J2ME vaatii yhden luokan, joka periytyy MIDlet luokasta. Siksi tätä on ollut luonnollista käyttää päätason Ohjaimena, joka luo muut Ohjaimet ja käynnistää ne. Päätason Ohjaimen huolehdittavana on luoda tarvittavat komponentit ja käynnistää aloitusnäkymä.

3.3.2 DataTypeIF



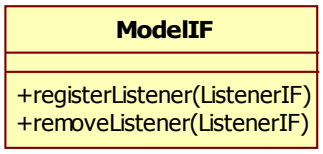
DataTypeIF-rajapinnan toteuttavat luokat toimivat ohjelmassa erilaisten laskutusperusteiden ilmentyminä. Nämä luokat kertovat itsestään Malli- ja Näkymä-luokille tarvittavat tiedot, jotta ne pystytään näyttämään käyttäjälle ja lähettämään eteenpäin mobiililaitteesta.

3.3.3 ListenerIF



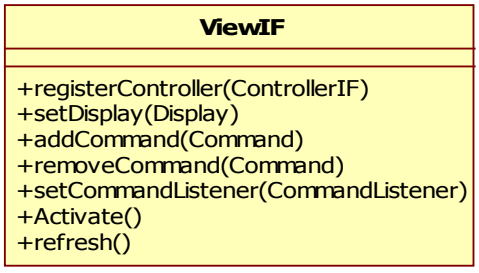
Tarkkailija-arkkitehtuuria toteuttamaan olen luonut ListenerIF-rajapinnan, jonka kautta Malli-luokat ilmoittaa Näkymä- ja Ohjain-luokille, että jokin Malli-luokan sisältämästä informaatiosta on muuttunut.

3.3.4 ModelIF



ModelIF-rajapinta pakottaa Mallin toteuttamaan Tarkkailija-rajapintaa, eli Kuuntelijoiden hallintaa. Tämä rajapinta on oleellinen ruudunpäivityksen nopeuttamiseksi. ModelIF-rajapinnan kautta on Kuuntelija-rajanpinnan toteuttavalla luokalla mahdollisuus saada tieto Malli-luokassa tapahtuvista muutoksista. Tämä on tärkeä Näkymä-luokkien päivityksen kannalta Jos Näkymä päivittää itseään jokaisella piirroskierroksella, sen toiminta hidastuu huomattavasti. Kuuntelemalla päivityksiä voidaan asettaa Näkymä piirtämään itseään uudestaan vain tietojen muuttuessa

3.3.5 ViewIF



Näkymillä on kaikille yhteisiä toimintoja ja näitä kuvastamaan olen luonut ViewIF-rajapinnan.

Tämän rajapinnan kautta Näkymä-luokat saavat Ohjain-luokiltaan tarvitsemansa komentonäppäimet, tiedon siitä, kuka komentoja kuuntelee ja viitteen Ohjain-luokistaan. Ohjain-luokka voi myös käskeä Näkymä-luokkiaan aktivoitumaan, eli tulemaan näkyväksi näkymäksi tai päivittämään itsensä.

3.4 Luokat ja niiden yhteistyö

Sovellukseen muodostui kuusi MVC-arkkitehtuurin mukaista luokkaa sekä yksi verkkoliikenteen mahdollistava luokka. Sovelluksen käyttöliittymä on tehty helpoksi käyttäjälle ja luokkien yhteistyö toteutettu mahdollisimman yksinkertaiseksi. Käynnistettäessä sovellus tarkastaa, onko sillä tarvittavat asetustiedot, joista käyttäjä voidaan tunnistaa ja matkalaskun tiedot voidaan lähettää eteenpäin. Mikäli tarvittavia

tietoja ei ole, sovellus näyttää ensimmäisenä näkymänä käyttäjälle asetus-näytön, johon käyttäjän tulee kirjata sovelluksen tarvitsemat tiedot. Mikäli asetustiedot löytyvät jo muistista, sovellus ei niitä kysy automaattisesti uudelleen. Halutessaan käyttäjä voi muuttaa näitä asetukset valikosta.

Kun sovelluksella on asetukset muistissa, se näyttää käynnistyessään automaattisesti matkalasku-osuuden ja käyttäjä pääse lisäämään matkansa tiedot, -kulut sekä päivärahavaateet. Annetut tiedot voidaan sovelluksen toimesta lähettää haluttuun sähköpostiosoitteeseen valmiissa formaatissa.

3.4.1 MIDlet luokka

Tämä luokka on pakollinen Java arkkitehtuurin vuoksi, J2ME järjestelmissä Java framework kutsuu sovellusta käynnistettäessä aluksi MIDlet-luokan aliluokkaa. MIDlet luokka tarjoaa metodit ohjelman käynnistämistä, sulkemista sekä keskeytystä varten. Tämä aliluokka luo sovelluksen tarvitsemat Controllerit sekä antaa näille viitteen matkapuhelimen näyttöön. Luokka myös antaa itsensä viitteenä eteenpäin, jotta sovelluksen sulkeminen onnistuisi helposti. J2ME toteutuksen sulkemista varten MIDlet luokka tarjoaa metodin, jolla voidaan varmistaa, että tarvittavat tiedot on tallennettu ja ohjelma on valmis sulkeutumaan. Tämän luokan kautta voidaan myös toteuttaa resurssien minimointi ohjelmasuorituksen katketessa esimerkiksi tulevan puhelun takia.

3.4.2 Matkalasku luokat

Näiden luokkien saumaton yhteistyö on sovelluksen kannalta erittäin tärkeä, sillä ne toteuttavat kaiken pakollisen toiminnallisuuden sovellukseen.

3.4.2.1 Controller

Tämä matkalaskuluokka on täysin vastuussa matkalaskua luodessa käytettävissä olevien komentojen toiminnallisuudesta. Luokka kutsuu tarvittavia funktioita Malli- ja Näkymä-luokista käyttäjän toimintojen perusteella. Mikäli käyttäjä haluaa esimerkiksi lisätä uuden matkan tiedot sovellukseen, hän painaa matkapuhelimesta ”Add...” näppäintä ja Ohjain- luokka vastaanottaa komennon ja hoitaa tarvittavat toiminnot käskyttämällä Näkymä-luokkaa pyytämään käyttäjältä tarvitsemansa tiedot. Käyttäjän syötteen jälkeen matkan tiedot lähetetään Malli-luokalle tallennusta varten.

Kuva 5: Poistettavan syötteen valinta sovelluksessa

3.4.2.2 Model

Matkalaskusovelluksen Malli-luokan tarkoituksena on säilöä käyttäjän syöttämät tiedot muistiin ja antaa pyydettäessä tiedot Näkymä-luokalle sekä verkkoliikennettä käyttävälle MIDP-luokalle. Tämä luokka on myös vastuussa tietojen tallentamisesta ohjelman sulkeutuessa seka vanhojen tietojen lataamisesta ohjelmaa käynnistettäessä.

Matkalaskun Malli-luokalle on annettu vastuu yksittäisten DataTypeIF-rajapinnan toteuttavien Tietotyyppien tallennuksesta.

3.4.2.3 View

Näkymä-luokka luodaan Ohjain-luokan toimesta ja sille asetetaan Malli-luokka tiedonhakua varten ja Ohjain-luokka komentojen tulkitsemista varten. Tämän luokan vastuulla on Malli-luokan tietojen näyttäminen käyttäjälle halutussa formaatissa. Kyseisestä Näkymä-luokasta tuli ehkä hieman liiankin monipuolinen lopullisessa sovelluksessa ja nämä toiminnallisuudet olisi voinut kirjoittaa myös useampaan Näkymä-luokkaan.

Kuva 6: Kuvakaappauksessa täytetty matkalasku

Lopulliseen sovellukseen Matkalasku-Näkymällä muodostui useita erilaisia tiloja, joiden avulla matkan tietoa pystytään luomaan, näyttämään, muokkaamaan ja poistamaan.

3.4.3 Asetusluokat

Asetuksien toiminnallisuus ei ollut ohjelmiston ydintoimintojen kannalta oleellinen. Käyttäjän tunnistamisen ja verkkoliikenteen tarvitsemien tietojen säilyttämisen ja keräämisen kannalta asetukset on kuitenkin pakollisia.

3.4.3.1 Controller & View

Ohjain- ja Näkymä-luokkien yhdistäminen on oiva keino yksinkertaisessa asetusnäkymässä. Asetusnäkymän tarkoituksena on saada käyttäjältä pakolliset tunnistetiedot ja verkkoasetukset, joten monimutkaista logiikkaa ei tarvita. Näiden kahden luokan eriyttäminen olisi hankaloittanut sovelluksen toteuttamista turhaan. Näkymä-luokan kannalta oleellinen ratkaisu luokkaa toteuttaessa oli toteuttaa myös ItemStateListener rajapinta, jonka avulla pystyttiin siirtämään tietoa ”reaaliaikaisesti”

Näkymä- ja Malli-luokkien välillä ja täten pitämään Malli-luokka jatkuvasti ajan tasalla käyttäjän tekemistä muutoksista. Ohjain-luokan vastuulle ei jää oikeastaan muuta kuin varmistaa Malli-luokalta, onko kaikki tiedot annettu, ja voidaanko jatkaa asetuksista eteenpäin varsinaisen sovelluksen käyttöön. Ohjain-luokka myös pyytää Malli-luokkaa palauttamaan aiemmat tietonsa, jos käyttäjä päättää perua jo antamansa uudet tiedot.

Kuva 7: Asetusten syöttäminen

3.4.3.2 Model

Koska asetuksiin ei normaaliolosuhteissa tarvitse tehdä muutosta ensimmäisen syöttökerran jälkeen, on Malli-luokalle annettu vastuu tietojen tallentamisesta matkapuhelimen muistiin ja niiden lataamisesta sovellusta käynnistettäessä. Asetus-malli osaa myös kysyttäessä kertoa, onko asetukset kohdallaan. Näin voidaan jo sovellusta käynnistettäessä päätellä, tuleeko käyttäjälle näyttää alkuun asetusnäkymä vai voiko hän suoraan mennä täyttämään matkalaskua.

3.4.4 Tietotyypit

Matkalaskusovellusta luotaessa on tarpeellista pohtia millaista tietoa sovellukseen tulee syöttää. Projektissa loin kolme erilaista tietotyyppiä, joiden avulla käyttäjä pystyy kertomaan sovellukselle, millaista matka-/kululaskua ollaan tekemässä.
Nämä tietotyypit ovat Päiväraha, Matka ja Kulu. Koska halusin pitää sovelluksen mahdollisimman yksinkertaisena, en luonut näistä omia J2ME:n CustomItem luokkia, vaan käytin olemassa olevia Item-luokkia, kuten TextField ja ChoiceGroup. Nämä luokat sidoin toisiinsa Vector-tietorakenteen avulla ja loin metodit, joiden kautta nämä pystytään enumeroimaan.

Tiedon varsinainen tulkitseminen käyttäjälle jätettiin täysin Näkymä-luokkien tehtäväksi. Esimerkiksi kun Näkymä-luokka haluaa näyttää käyttäjälle hänen antamansa syötteet, se muuttaa TextField luokat StringItemeiksi, joita ei pystytä suoraan editoimaan.

Tietotyyppien sisällä on ohjelmiston kannalta tärkeimmät tiedot. Näihin luokkiin on kirjoitettu tieto matkan alkupisteestä, päämäärästä, kululaskujen summista ja siitä mihin tarkoitukseen hyödyke on hankittu. Näihin luokkiin olisi voitu myös kirjoittaa valmiiksi valittavia tietoja esimerkiksi lähtö- ja saapumispisteistä mutta yksinkertaisuuden vuoksi nämä jätettiin pois. Matkalasku-Tietotyypille annettiin kuitenkin optiolista matkustusvälineen valintaa varten.

3.4.5 Verkkoliikenne

Sovelluksen verkkoliikenne hoidetaan TCP/IP yhteyden ylitse SMTP protokollaa käyttäen. Tätä tarkoitusta varten on luotu EmailClient niminen luokka, jonka tehtäviin kuuluu kysellä sekä matkalaskun että asetuksien Malli-luokilta tarvittavat tiedot ja muokata niistä valmis teksti lähetettäväksi haluttuun sähköpostiosoitteeseen. Sähköpostin lähetys tapahtuu omassa säikeessään ja luokka informoi käyttäjää lähetystapahtuman etenemisestä.

```
  PINE 4.53   MESSAGE TEXT

Date: Thu May 14 16:11:32 UTC 2009
From: Heikkilä Juha
To:
Subject: Travelexpense invoice - Heikkilä Juha

From: Tampere to Helsinki with Own Car for 185 km while carrying 1 passengers.
Lunch with customer : 85
```

Kuva 8: Ohjelman lähettämä valmis sähköposti

Käyttäjän reaaliaikainen informointi on tärkeää käyttöliittymän kannalta. Tämä informoi käyttäjää myös ohjelman toimivuudesta. Mikäli ohjelma vaikuttaa käyttäjästä

siltä, ettei sovellus enää toimi, käyttäjä todennäköisesti yrittää lopettaa lukkiutuneelta vaikuttavan sovelluksen. Tämä puolestaan saattaa aiheuttaa tallentamattomien tietojen menettämisen tai sähköpostin lähettämisen keskeytymisen.

Tämä apuluokka saa Asetus-mallilta yhteystiedot palvelimeen sekä sähköpostiosoitteet lähettäjälle ja vastaanottajalle. Näitä tietoja hyväksikäyttäen ohjelma osaa muodostaa SMTP protokollan vaatimat tiedot oikeanlaiseen formaattiin. Luokka saa myös Matkalasku-mallilta tietoja käyttäjän syötteistä ja osaa muodostaa niistä selkokielisen listauksen palkanlaskennan käyttöön.

4 Testaus

MVC-arkkitehtuuri helpottaa ohjelman toiminnallisuuden testaamista, koska erilliset komponentit voidaan testata toisistaan riippumatta. Koska komponentit on jaettu useampaan osaan, voidaan yksittäisiä osia testata yksikkötesteillä helpommin, koska testeistä pois suljetut osat on helppo korvata Mock Objekteilla. Testien tulokset eivät silloin ole riippuvaisia reaalitiedoista, vaan ohjelman perustiedot ovat aina samat eikä testitapauksia varten tarvitse asettaa alkutietoja joka tapauksen välissä.

5 Tulokset, arviointi ja päätelmät

Tulokset

Kokonaisuudessaan projekti onnistui mielestäni hyvin. Ohjelmiston kehitys on vaiheessa, jossa perustoiminnallisuus on toteutettuna ja jatkokehitys voidaan aloittaa. Itselleni projekti oli hyvin opettavainen ohjelmistokehityksen kokonaiskuvan hahmottamisen kannalta ja paransi tietämystäni paitsi MVC-arkkitehtuurista, myös muista ohjelmistoarkkitehtuureista sekä niiden käytöstä. Ohjelmisto toteutti kaikki vaatimusmäärittelyssä eritellyt primääriset toiminnot sekä suurimman osan sekundäärisistä toiminnoista. Ohjelmistoa kokeillessa huomasin ruudunpäivityksen hidastelevan enemmän kuin olisi oletettavaa tämänkaltaisessa sovelluksessa. Ohjelmisto ei teoriassa ole raskas, mutta jatkuvat luokkien väliset kyselyt hidastavat ohjelman suoritusta ja luokkajaottelu luo ylimääräistä rasitetta muisti- ja prosessorikapasiteetille. Tästä ilmiöstä voisi päästä eroon optimoimalla ohjelmakoodia entisestään.

Arviointi

MVC-Arkkitehtuuri soveltuu hyvin laajojen graafisten ohjelmistojen suunnitteluun ja toteutuksiin. Pienemmissä projekteissa se luo turhaa kuormitusta suunnitteluun sekä ohjelmatoteutukseen. Näin ollen Mobiililaitteiden ohjelmistoissa MVC-arkkitehtuuri ei ole parhaimmillaan. Mobiililaitteet asettavat rajoituksia sekä prosessoritehon että muistitilan kannalta ja MVC-arkkitehtuuri luo ylimääräistä rasitusta kumpaankin. Tilanne ei nykyisillä matkapuhelimilla ole kovin vakava, sillä niiden resurssit ovat kehityksen myötä kasvaneet huomattavasti, mutta ohjelmistot pitäisi mielestäni aina suunnitella toimimaan mahdollisimman monessa eri laiteympäristössä, mukaanlukien vanhemmat mallit.

Opinnäytetyön yhtenä tavoitteena oli sovelluksen toteuttaminen helposti integroitavaksi ja ohjelmakoodin uudelleen käytettävyys. Tässä tavoitteessa onnistuin myös mielestäni hyvin. Ohjelmakoodi on itsenäisesti toimiva ja integrointi onnistuu siksi helposti.

Opinnäytetyönä toteutetussa kulu- ja matkalasku sovelluksessa MVC-arkkitehtuurista ei pystytty hyödyntämään sen parhaita puolia. Ohjelmistoprojekti oli liian pieni hyödyntääkseen täysin MVC-Arkkitehtuurin tuomia apuja ohjelmointiin. Olin itse vastuussa kaikkien komponenttien rakentamisesta, joten graafinen ulkoasu jäi kovin askeettiseksi ja Malli-luokkien tiedot jäivät optimoimatta. Siirrettävyys näkyi kuitenkin ohjelmistosta ja siitä tuli helposti muokattava, joten jatkokehitystä ajatellen MVC-arkkitehtuurin pakottama komponenttien erottaminen oli hyödyllinen.

MVC-Arkkitehtuurin suurimpana haittana pidetään sen tuomaa monimutkaisuutta ohjelmistoon ja tämän huomasin myös itse ohjelmistoa rakentaessani. Monen komponentin kohdalla olisi päässyt huomattavasti pienemmällä vaivalla, mikäli sen toiminnallisuutta ei olisi eritelty useampaan luokkaan. Uskon kuitenkin, että tämä toimintojen erittely helpottaa jatkokehitystä monessa kohdassa. Tämän ansiosta ohjelmasta saadaan helposti muokattua yrityksen ulkoasun mukainen vaihtamalla Näkymän käyttämiä Item-luokkia omiksi CustomItem-luokiksi.

Kokonaisuuden testaaminen jäi hieman vaillinaiseksi. Mikäli ohjelmistoa aletaan räätälöidä osaksi suurempaa kokonaisuutta tai kehittää siitä markkinoille soveltuvaa mallia, niin testaamiseen on syytä panostaa huomattavasti enemmän.

Ohjelman ulkonäkö jäi askeettiseksi, koska se käyttää pelkästään Javan sisältämiä valmiita graafisia luokkia eikä toteuta grafiikoita itse.

Päätelmät

Mobiiliohjelmointiin suositellaan mahdollisimman vähän eri luokkien käyttämistä, mutta MVC-arkkitehtuuri pakottaa yksittäisen luokan jakamisen useampaan osaan, ja täten kapinoi kyseistä sääntöä vastaan. Erillisten luokkien käyttämistä ei suosita sen takia, että jokainen ylimääräinen luokka luo hieman ylimääräistä muistinkäyttöä. Vaikka tämä muistinkäyttö ei olekaan kovin suurta, saattaa se muodostaa ongelmia mobiileissa laitteissa, missä resurssit ovat rajalliset ja jokainen säästetty tavu on merkittävä.

Mobiiliohjelmoinnissa MVC-arkkitehtuurin käyttöä kannattaa harkita tarkkaan sen luoman ylimääräisen resurssikulutuksen vuoksi. Vaikkakin MVC-arkkitehtuuri kuluttaakin resursseja mobiililaitteissa kohtuullisen paljon, on se modulaarisuutensa vuoksi silti varteenotettava vaihtoehto. Mobiiliteknologia kehittyy huimaa vauhtia, kasvattaen mobiililaitteiden resurssikapasiteettia ja siten tehden MVC-arkkitehtuurista entistä tehokkaamman arkkitehtuurimallin mobiililaitteidenkin ohjelmistokehitykseen.

6 Jatkokehitys

Näin yksinkertainen matka- ja kululaskusovellus ei sellaisenaan ole helposti markkinoitava tuote, mutta MVC-arkkitehtuurin ansiosta sitä on helppo liittää muihin tuotteisiin tai käyttää yhtenä osana asiakkaan tarpeisiin räätälöityä kokonaisuutta.

Ohjelmiston jatkokehityksen kannalta MVC-arkkitehtuuri luo hyvän pohjan helppoon kehitykseen. Arkkitehtuurista johtuvan komponentin jakautumisen ansiosta voidaan jokaista komponentin palasta parantaa osa-alueen asiantuntioiden avustuksella. Esimerkiksi Näkymää voi hioa käyttöliittymäsuunnittelijat, jotka ymmärtävät ihmisen ja tietokoneen vuorovaikutuksen parhaiten.

Ohjelmasta yhä puuttumaan jäi luotettava käyttäjäntunnistaminen. Tähän toimintaan suunnittelin alunperin puhelimeen perustuvaa tunnistusta, mutta miettimieni matkapuhelintunnisteiden, kuten IMEI tai IMSI tai puhelinnumero, saaminen mobiililaitteesta ei kuulu J2ME-API:n. Ainoaksi varteenotettavaksi vaihtoehdoksi jäi yksilöity tunnistenumero sovellukseen, mutta tämä olisi sitonut ohjelmiston käyttäjän yksittäiseen puhelimeen, eikä siten olisi ollut kovin toimiva ratkaisu. Koska ohjelmisto haluttiin pitää mahdollisimman yksinkertaisena ja omatoimisena, kirjautumispalvelin jäi pois mahdollisista tunnistautumiskeinoista.

Kilosoft Oy:n tarkoituksena oli liittää Kulu- ja Matkalasku-sovellus yrityksen toiseen mobiilisovellukseen ja tätä tarkoitusta varten Kulu- ja Matkalasku-sovelluksen Näkymä-luokka tulisi muuttaa omia visuaalisia komponentteja käyttäväksi ja sähköpostinlähetyksen rinnalle suunnitella SOAP viestejä käyttävä lähetystoiminto. Tarvittavat muutokset olisi kohtuullisen yksinkertaista toteuttaa valmiin sovelluksen lisäksi.

Ohjelmisto toteutettiin käyttämään HTML-pohjaista sähköpostinlähetystä mutta tämä olisi hyvä korvata tulevaisuudessa SOAP muotoisina XML viesteinä. Tämä helpottaisi ja nopeuttaisi viestejä vastaanottavan sovelluksen toteuttamista valmiiden komponenttien ansiosta. Sovelluksen tämän hetkinen tiedonsiirto edellyttää vastaanottajaksi ihmistä mutta SOAP protokollan avulla voitaisiin helposti kehittää

vastaanottajaksi sovellus joka pystyisi siirtämään tiedot suoraan laskutusjärjestelmään. Koska vastaanottava sovellus todennäköisesti toimisi resurrseiltaan tehokkaammassa ympäristössä, olisi sinne mahdollista toteuttaa myös tästä sovelluksesta uupuva mahdollisuus laskea päivärahat.

Mobiililaitteiden heikkoja resursseja hyväksikäyttäen vuosittain muuttuvat päivärahojen arvot sekä suurempi kompleksisuus päivärahojen tuomisessa mobiililaitteeseen teki tämän liian haasteelliseksi toteuttaa.

Lähteet

Aalto, Juha-Markus; Aalto, Ari; Jaakso, Ari; Vättö, Kimmo 1999. Tried & True Object Development Industry-Proven Approaches with UML. New York, USA: Cambridge University Press

Design Patterns in Symbian 2007. [online] [viitattu 26.11.2008] http://wiki.forum.nokia.com/index.php/Design_Patterns_in_Symbian

Java BluePrints – J2EE Patterns n.d. [online] [viitattu 8.12.2008] http://java.sun.com/blueprints/patterns/MVC-detailed.html

Koskimies, Kai; Mikkonen, Tommi 2005. Ohjelmistoarkkitehtuurit. Helsinki: Talentum Media Oy

Model – View – Controller 2008. [online] [viitattu 8.12.2008] http://msdn.microsoft.com/en-us/library/ms978748.aspx

Origins of Software Architecture Study 2008. [online] [viitattu 8.12.2008] http://www.sei.cmu.edu/architecture/roots.html

Yuan, Michael Juntao 2004. Enterprise J2ME Developing Mobile Java Applications. New Jersey: Pearson Education inc